A Survey of Data Models

Jim Kowalkowski and Marc Paterno
FNAL/CD/CEPA
Revision 1.0

Contents

1 Introduction

In this document we present a general description of an *event data model*, including the purpose and common features of such. We present a brief survey of some existing event data models. We also present some lessons learned from the design, implementation, and review of these event data models.

1.1 What is an Event Data Model?

An Event Data Model (EDM) is a set of software components[1] which provide a mechanism for managing data related to an "event" (*i.e.* a collision)[2] within a program. An EDM is not merely a persistency mechanism, nor is it an input/output mechanism a file format — although it is related to these things. We shall discuss only *object-oriented* EDMs, that is, models in which the "event data" consists of a group of interacting objects. In one case, we shall discuss an EDM which have features that allow procedural languages to access event data.

[1] More formally, the software components are the *implementation* of the model, which is the set of concepts represented by the implementation. But in this document, we're not going to be that formal.

[2] Throughout this document, we use nomenclature common to collider experiments. We do this because it is the domain with which we are most familiar. We trust our colleagues in the fixed target field can translate terms appropriately.

An EDM is different from a *framework*, which provides a mechanism for constructing programs (typically triggering, filtering, reconstruction, analysis, and event display) programs from independent *modules*. Often, an EDM has no knowledge of (or dependence upon) a framework, and a framework has little or no knowledge of the EDM.

Most EDMs consist of a core of classes that form a basic infrastructure, and a large set of classes which represent the "physics objects" of the system. The infrastructure classes should be completed early in the development cycle. In contrast, the set of physics objects is infinitely extensible, and is in continuous flux.

The i/o system should use the EDM, viewing data objects in an abstract fashion, and performing i/o for the user without requiring knowledge of the specific physics objects in the EDM.

1.2 Why is an EDM Useful?

Use of a clearly defined EDM provides several benefits for an experiment.

- It allows for independence of framework modules. This is achieved by having the framework modules communicate *only* through the EDM. Independent modules can be developed, tested, and upgraded independently, which is critical for any substantial body of code that is to be maintained for several (or many) years.
- It allows for independent development of data objects and algorithms when such development is possible. It enhances the orthogonality in design—which leads to more maintainable code.
- It can isolate users from need to interact with persistency mechanism. The core code of the EDM can handle routine manipulations, allowing physicists to concentrate on the important business, rather than busying themselves with repetitive coding.
- It can isolate users from input/output mechanism. A well-defined EDM allows the framework to handle the business of handling files (or other data sources), and reading and writing events.
- It can isolate users from the details of the file format. Changes in the file format do not impact users.

1.3 The Common Features of EDMs

Some features are shared by (nearly) all EDMs. Here we describe these features in general terms; in §3, we describe each in more detail, in the context of several example experiments.

Each EDM has an "event" class, which represents all the data associated with a single collision. The "event" is generally some sort of collection of contained objects, although the nature of the collection can vary widely. Generally, the "event" is responsible for managing the lifetime of its contents.

Each EDM has many classes representing various "pieces" of an event. Some of these "pieces" are "raw data" (*e.g.* ADC counts from a calorimeter cell), others are "processed data" (*e.g.* a track found in tracking device), and others are formed from associations of either or both of these items (*e.g.* an electron candidate formed from a group of calorimeter hits and a track).

Each EDM contains navigation classes. These include the classes used to find specific items within the event (*e.g.* all the tracks found by a specific algorithm) as well as the classes used to

associate items with other items in an event (*e.g.* this muon candidate is associated with that track).

Each EDM contains metadata classes, which *describe* the event data and its processing, rather than representing event data. Some examples are run and event numbers, triggers satisfied by an event, and identification of which algorithms were responsible for the creation of a specific event datum.

2 Common Needs

The features of an EDM must fit the needs of the experiment using that EDM. There are some needs that seem common to all experiments; in §3, we identify these common needs. In a later section we describe the EDM features that satisfy these needs.

An EDM must support flexibility in program design. Probably the most important manner in which an EDM supports flexibility is by providing the means by which more than one algorithm is allowed to produce a given kind of output. The EDM must supporting holding multiple instances of a given type of output, and must also support distinguishing one from another. For example, an experiment may want to produce jets by both cone algorithms and K_T algorithms. It is necessary to distinguish between the jets made by different algorithms. Furthermore, a single algorithm can be configured with different parameters; it is necessary to distinguish between $R = 0.7$ cone jets and $R = 0.4$ cone jets. The EDM must provide a mechanism to unambiguously identify the origin of each jet.

An EDM must support extensibility. Many different types of reconstructed objects need to be stored in the event. All of these types make up "the EDM". However, not all of these types can be provided in the core package that makes up the EDM. There is often a continuous need to add new types, needed to support new physics work, through the lifetime of an experiment. It hopeless (and counter-productive) to pretend that all the types can be determined at the beginning of the experiment. The EDM must make it possible for new types to be added without breaking existing code, and without requiring significant modification of the EDM. This allows the EDM to grow incrementally, as user demands guide the growth.

The EDM must support modification. It should be expected that early versions of most of the classes defined to represent event data objects shall be less than perfect. As new information is gained about the needs of users, it must be possible to update the classes without rendering unusable those data written with the older definitions. Such *schema evolution* must be built into the EDM.

3 A Survey of Selected EDMs

In this section, we present a survey of several existing EDMs. The choice of experiments surveyed is limited to those experiments with which the authors are reasonably familiar. Possible future revisions of this document may contain information from additional experiments.

In the notation below, **CDF** and **DØ** refer to the event models of those experiments; **Gaudi** refers to the event model under development for LHCb[3], **MB** refers to the event model of the MiniBooNE experiment, and **MB(F)** refers to the Fortran interface to the MiniBooNE event model.

[3]Note that Gaudi refers to the status of the LHCb implementation of the EDM portion of the Gaudi framework, as of Sept. 2001.

3.1 The Event Class

3.1.1 How does the user get access to the Event?

CDF Passed into function calls, also available in a global variable.
DØ Passed into function calls.
Gaudi Query for search of global singleton registry.
MB Passed into function calls.
MB(F) Globally available.

3.1.2 Can there be more than one Event available at one time?

CDF Yes, but use of the global would cause trouble.
DØ Yes.
Gaudi Yes, but it is unclear how to access them through the global store.
MB Yes.
MB(F) No.

3.1.3 What is the basic style of the interface?

The question here is how the event appears to the user; what sort of interaction methods are provided?

CDF Iteration over a container of abstract nodes.
DØ "Database" with a variety of defined queries, typesafe.
Gaudi Filesystem-style hierarchy of named abstract nodes.
MB Associative array of nodes, typesafe.
MB(F) Subroutine calls to load common blocks.

3.1.4 How do we add data to the event?

CDF Append to event; ownership passed to Event (by design), no copying.
DØ Insert into event; ownership passed to Event (by design), no copying.
Gaudi Insert into event; ownership is passed to Event (by convention), no copying.
MB Insert into event; ownership passed to Event (by design), no copying.
MB(F) Subroutine call to copy from common block, into an object which is inserted into the event.

3.1.5 Are inserted items immutable?

CDF Yes, except that some items can "grow".
DØ Yes.
Gaudi No.
MB *under development*
MB(F) *under development*

We believe that having an Event class is important because the "event" is an important feature of the problem domain—it models reality.

Having an Event passed into the functions which shall modify it, or which read it, decreases the impact of *side-effects*. Side-effects can induce a variety of problems:

- they produces coupling between conceptually distant parts of the code, making the code more difficult to understand;
- they hinders optimizations;
- they hinders use of multiple threads of execution;
- they makes testing more difficult.

If the interface to the Event is insufficient, *e.g.* too low level, users shall invent their own, perhaps on a subsystem-by-subsystem basis. This is terrible, as it leads to a different high-level interface for each subsystem. Too rich an interface can also be a problem—many users shall use the "simplest thing possible". The simplest *correct* interface should be presented. For example, experience has shown when an "easy but somewhat dangerous" interface and a "safe but slightly less easy" interface are both presented, most users shall choose the easy road. Experience has also shown that eventually the danger shall bite, and recovery can be painful.

A well-designed Event interface provides helpful uniformity of access to the different items stored in the Event. This uniformity is valuable because it gives users less learning to do (both for designers of additions to the EDM and for end-users of the EDM).

We believe it is important for an EDM to always make clear ownership of all resources (primarily, but perhaps not only, the ownership of objects). Smart pointers, handles, *etc.*, facilitate correct use; reliance upon convention has often caused severe problems—memory corruption and memory leaks being the most common examples.

We believe the immutability of the elements in an Event is important. Experiences has shown that users who have violated this immutability have caused weeks of (unnecessary) debugging effort for other users. The problems are sometimes not recognized until some "interesting" feature of the data is investigated—for a lengthy period of time—leading to the discovery that the "feature" is in fact an error in processing. In some cases, reproducibility of an analysis is destroyed by modification of elements in the Event. We also believe that some modifications of the elements in an Event (such as CDF's ability to add to collections in a controlled fashion) are both safe and valuable.

3.2 "Pieces"

What is required for a class to become part of the EDM—that is, for instances of this class to be stored in an Event?

3.2.1 What requirements are imposed?

CDF Inheritance from *TObject* via *StorableObject*. Requires CDF macro, to write some of the interface required by ROOT.

DØ Inheritance from *AbsChunk*. Requires DØ macro, to write some of the interface required by DOOM. Requires possession of identifying information (IDs of parents, IDs of algorithm used for generation, IDs for "environment".

Gaudi Inheritance from *DataObject*. Requires ability to return a globally unique integer ID for the class.

MB Any class that is CopyConstructible. Should be a POD; current usage of ROOT violates this.

MB(F) Any (properly padded) common block; no strings are allowed.

3.2.2 Can we have more than one of each type active?

CDF Yes.
DØ Yes.
Gaudi Yes.
MB Yes.
MB(F) No.

Experience has shown that it is necessary to be able to run the same algorithm with different configurations multiple times for the same event, and to save the results from each invocation of the algorithm. In the next section, we concentrate on the importance of being able to unambigiously distinguish between the results of these invocations.

It is important for the design of these classes to have the compiler enforce as many of the "design rules" of the EDM as possible. It is much worse for these "design rules" to exist only as a check list for the various users to obey. Experience has shown that enforcing such rules is extremely difficult, and grows more difficult as time progresses and as the size of the project expands.

Experience has shown that the more "data-like" these objects are they easier they are to deal with in many ways.

- It is easier to map struct-like data (rather than rich C++ classes) to a persistent format.
- It is easier to manipulate such objects from other languages.
- It is easier to manipulate such objects in a variety of analysis tools.
- It is easier to handle schema evolution for simple data-like objects.

We do *not* mean to say that an object-oriented design is less good than a procedural or simply object-centric design. The rich C++ interface to the data objects can be provided by classes which encapsulate the simple data-like objects suggested here.

3.3 Searching

3.3.1 How are instances of objects within the event labeled?

CDF Unique object ID, configuration parameter set ID, descriptive string, class version, and class name.
DØ Unique object ID, configuration parameter set ID, parent object IDs, and geometry and calibration IDs, and string labels.
Gaudi Class ID, descriptive string with hierarchical path.
MB Descriptive string and class name.
MB(F) Descriptive string.

3.3.2 How does a user locate objects in the event?

CDF Custom iterators with optional selectors specifying a combination of labeling information. Find by class ID.
DØ Find with user specified criteria or specific labeling information. Muliple objects returned.
Gaudi Find with string path information.
MB Find with class name/descriptive string. Single object returned
MB(F) Find with descriptive string. Single object returned

3.3.3 What is returned from a search for objects in the event?

CDF Custom iterator that allow constant access to the object they refer to and traversal to next object.
DØ A collection of handles that allow constant access to the objects.
Gaudi A bare pointer to the base class object or to the object itself. Modification is dicouraged by convention only.
MB A contant pointer to the object
MB(F) A populated common block

3.3.3.1 How are multiple matches handled?

CDF The returned iterator moves through the matches.
DØ A collection of matches in returned.
Gaudi Not Applicable, Atlas is preparing iterator handle mechanism.
MB No multiple matches implemented yet.
MB(F) No multiple matches allowed.

Experience has shown that richness in labeling is important. Users shall encounter needs for organizations of the data that are not anticipated in early designs; a rich labeling (and searching) system allows them to obtain such views without modification to the core EDM. A too-simple, fixed view of the data does not suffice. A single hierarchical view does not suffice.

We caution against lack of precision in the specification of what "pieces" of the Event are to be returned as the result of some query. One should never expect there shall be only one instance of any class; experience has shown that even if this is true today, it shall not remain so forever. Concepts such as "the best version" or "the most recent version" are too imprecise to be safe, and have caused troubles when used. Returning "the default version" is dangerous because it can easily lead to irreproducibility. Experience has shown that later steps in reconstruction—when they rely on "default versions" from earlier steps—can become dependent upon the *order* in which previous reconstructions steps are done, because the meaning of "default" can easily change, unbeknownst to the user.

3.4 Associations

3.4.1 How are association supported?

CDF Using special link classes that are converted from pointer to id and back by the system. Link classes exist for objects with collection assocations.
DØ Special link classes for the persistent and pointer representations. Semi-automated conversion process for saving and restoring. Link classes exist for objects with collection assocations.
Gaudi Special class that link to a *DataObject* or vector of *DataObject*s.
MB Currently no infrastructure support.
MB(F) Not available.

3.4.2 What restrictions are in place?

CDF No bare pointers allowed by convention. References to object directly registered in the event. Only references to already recorded objects are allowed (one way, new to older)

DØ No bare pointers from one object to another by convention. References to object directly in the event or objects held within arrays that are held in the event. Same one way restrictions as CDF

Gaudi Desire the use of the link classes only. References to objects derived from their base class. Similar one way restructions.

MB Currently no infrastructure support.

MB(F) Not available.

3.4.3 How are N-way associations supported?

CDF Separate object in the event ties the objects together using the support link classes.

DØ Separate object in the event ties the objects together using the support link classes.

Gaudi unknown

MB Not available.

MB(F) Not available.

Experience shows that it is not feasible to store every type of reconstructed object as a top-level "piece" of the event. For example, small objects such as individual tracking chamber "hits" are not suitable candidates as top-level event "pieces", because of the cost of retaining provenance information about each object. For this reason, most "pieces" of an event turn out to be collections of objects (hits, tracks, jets, *etc.*), and in many instances these objects within the "pieces" need to be referenced from outside the "piece". An EDM must provide a flexible mechanism for referencing such objects.

None of the EDMs surveyed directly support the creation of associations between groups of objects which do not otherwise know about each other. The addition of such a facility would be beneficial.[4]

3.5 Persistency

3.5.1 What requirements are imposed?

Two types of classes can be destinguished here: the ones that are appended directly to the event, and the ones that are children of the directly attached objects.

CDF Macros to describe the directly attached classes. Streamers to serialize/deserialize the data in the object. Indirect inheritance from TObject through the StorableObject base class.

DØ Indirect inheritance from the D0Object class from AbsChunk for directly attached classes. Macros that write methods required by D0OM.

Gaudi All data must be publicly available or provide set/get methods to adjust the state of the object externally.

MB Currently require the use of ROOT ClassDef macros.

MB(F) A C struct be maintained that maps onto the common block.

[4] It is our understanding that the CLEO EDM provides class templates to support such associations.

3.5.2 What restriction are in place?

CDF Not an automated procedure, so anything that can be streamed out is allowed.
DØ Classes (main and the children) must be parsable by d0omCINT.
Gaudi None.
MB Simple elementary types and array of these types.
MB(F) C++ is not allowed — this is Fortran.

3.5.3 What features does it support?

CDF Streamer / BLOB.
DØ Isolated from the persistency mechanism. Multiple backends can be supplied.
Gaudi Isolated from the persistency mechanism.
MB ROOT dictionary.
MB(F) ROOT dictionary.

3.5.4 What is the file format?

CDF ROOT.
DØ DSPACK is the standard backend, others can be supported.
Gaudi Objectivity and ROOT (BLOBs) are available.
MB ROOT.
MB(F) ROOT.

3.5.5 How is schema evolution supported?

CDF Encoded as if statements in the streamers.
DØ Automated by D0OM, uses a data dictionary merge.
Gaudi Encoded as if statements in the converters.
MB ROOT data dictionary merge currently.
MB(F) Same as Above MB.

3.5.6 What is the object translation mechanism?

CDF Hand written code to write object's data into the ROOT buffer. Sometimes time consuming, highly compact. The transient representation typically differs significantly from the persistent form. Transation managed through *postRead*, *preWrite* methods of the object.
DØ Persistency is automated by a data dictionary. Automatically generated code per class copies data to the DSPACK bank structure. Rarely used activate/deactivate can do simple transient mapping. Frequently used to resolve references between objects.
Gaudi Converter external to the class read state out into the persistency package buffers. Example is to copy the data objects into objectivity objects, and then write the those objects.
MB ROOT data dictionary.
MB(F) Copy of the common block to C++ objects, then to the ROOT dictionary as above.

In §3.2 we present some arguments for simple, data-like "pieces"; some of these relate to ease of support for persistency.

We observe that each experiment has chosen a single persistent format. It does not seem that the effort that would be required to obtain independence from a persistency toolkit is worthwhile, because of the amount of effort it would take to do it well. While our inclination is to say that the flexibility of being able to easily move to a different persistence mechanism is important, our observation has been that the toolkit in wide use—ROOT—is too intrusive to make this feasible. For example, users want to tune parameters for efficiency; this require detailed knowledge of exactly *how* ROOT works with buffers, *etc.*

Experience has shown that it is important for the persistency mechanism to have knowledge about the EDM, but not for the EDM to have knowledge of the persistence mechanism. Inversion of this relationship has caused significant trouble.

We have found that it is crucial to allow for hand-written "streamers" for schema evolution. Automatic mapping for simple schema evolution is helpful, but experience has shown that is it not sufficient to handle all cases.